Hydro-dynamics in CUDA
Performance optimizations in MOHID

Author
Jonathan van der Wielen

Date
2011-09-20

Last modified
2011-11-04

Version
1.0
### Version Control

<table>
<thead>
<tr>
<th>Version</th>
<th>Editor</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Jonathan</td>
<td>First draft of this document.</td>
</tr>
<tr>
<td>1.0</td>
<td>Jonathan</td>
<td>Added flow chart diagrams, conclusion, improved chapter 1</td>
</tr>
</tbody>
</table>
Introduction
This document describes the integration of CUDA into MOHID, a collection of hydro-dynamical models.

Chapter one documents how the Thomas algorithm has been integrated into MOHID. It also describes the used binding between FORTRAN and C.
1 Thomas algorithm: CUDA in FORTRAN example

The Thomas algorithm is a tri-diagonal solver for matrices. It is used in MOHID to calculate diffusion of concentration, and to calculate velocity. It is a performance bottleneck; it can take up to 10% of the execution time in a MOHID run, depending on the configuration. Therefore the Thomas algorithm has been translated to CUDA as a proof of concept, to see if there are any performance gains. This chapter documents the necessary steps to run CUDA code in FORTRAN, and documents the optimizations of the Thomas algorithm.

1.1 Parallelization

The Thomas algorithm is suitable to run on GPU, since the algorithm loops over one dimension in a 2D or 3D grid. For example in the case of a 3D grid, the algorithm loops over the Z dimension for all X columns and Y rows. Each [X, Y] is calculated independently, so a 2D grid of X by Y threads can be started. In the same manner a [Y, Z] grid can be started when calculating for the X dimension.

1.2 Parameters

The call to Thomas gets the following parameters:

- ILB, IUB, JLB, JUB, KLB, KUB. These are the lower and upper bounds of the calculation area of I, J and K, where I = X, J = Y and K = Z.
- D, E, F, TI. These are the coefficients and the independent term used to solve the algorithm.
- RES. This is the matrix where the resolution of the algorithm is stored.

1.3 Process flow of Thomas in MOHID

Before integrating Thomas in CUDA into MOHID, it is important to know which modules use the Thomas algorithm. Currently there are three Thomas variants: Thomas_2D, Thomas_3D and ThomasZ.

Thomas_2D is not converted to CUDA. Thomas_3D runs the Thomas algorithm for the X and Y dimension and ThomasZ runs the Thomas algorithm for the Z dimension.

**TABLE 1-1** shows the process flow of the modules that call ThomasZ and Thomas_3D.

```plaintext
// MohidWater: Main
ModifyMohidWater
  DoOneTimestep
    // ModuleModel
    RunModel
      RunOneModel
        // ModuleWaterProperties => RES
        WaterProperties_Evolution
          Advection_Diffusion_Processes
            // ModuleAdvectionDiffusion=> D, E, F, TI
            AdvectionDiffusion
              AdvectionDiffusionIteration
                ThomasZ
                Thomas_3D
                HorizontalAdvection
                  Thomas_3D
                // ModuleFreeVerticalMovement => D, E, F, TI
                FreeVerticalMovements_Processes
                  Modify_FreeVerticalMovement
```
It is important to realize where the parameters for the algorithm are allocated and de-allocated since some optimizations for CUDA require a different type of memory allocation. The grey lines with D, E, F, etc. show which parameters are allocated and de-allocated in which module.

1.4 ModuleCuda (FORTRAN)
ModuleCuda handles the communication between MOHID and CUDA. It has interfaces for the C binding. Each model that is created initializes one ModuleCuda instance, as is common in MOHID. Each module that uses CUDA associates the ModuleCuda instance. This is done by passing an ObjCudaID around.

ModuleCuda contains both initializing and killing subroutines for CUDA as all the Thomas subroutines. In the future these could be separated but for now it is sufficient to keep them together.

The FORTRAN – C binding is described extensively in §1.7.

1.5 Project CudaWrapper (C++)
The C++ project CudaWrapper is a wrapper to make CUDA calls easier. It contains error handling for CUDA API calls, methods to allocate and free page-locked memory, methods to initialize and kill
CUDA and methods for matrix transposing on the device. The matrix transposing is described in the programming guide\(^1\). Setting up a project like CudaWrapper or CudaThomas is also described in the programming guide.

CudaWrapper also contains a number of C binding methods (see §1.7.1).

## 1.6 Project CudaThomas (C++)

The CudaThomas project contains the class Thomas. An instance of Thomas is created for each ModuleCuda instance, if InitializeThomas is called. Each instance gets the ID of ModuleCuda, so it can be found again when needed. The Thomas class contains a static map of Thomas instances.

### 1.6.1 Creating and initializing an instance

To create a Thomas instance, CreateInstance should be called with the CudaID and the dimensions of the model as parameters. CreateInstance checks if an instance with the given ID already exists. If so, nothing is done. If the instance does not exist yet, it is created and initialized.

The Initialize method allocates memory on the device for the W, G, D, E, F, TI and Res matrices. The Thomas instance can solve Thomas for all dimensions. Because the X dimension requires a matrix transposing, separate matrices are allocated for this dimension. §1.9.2 explains why this is necessary.

### 1.6.2 Solving the Thomas algorithm

The Thomas algorithm can currently be solved for a 3D matrix for the X, Y or Z dimension. Three methods are used: SolveThomasX, SolveThomasY and SolveThomasZ. These methods all have the following process flow:

- Copy D, E, F, TI and Res to the device
- Execute DevThomas on device
- Copy Res back to the host

W and G do not need to be copied to the device, since they are temporary matrices only used by the kernel. Res needs to be copied to the device because the boundary values need to be preserved. This could be avoided by copying the boundary values to temporary matrices on the host, and copying them back into Res after solving the algorithm.

Solving the Thomas algorithm for the X or Z dimension requires calling the kernel DevThomasIK. Solving the algorithm for the Y dimension requires calling the kernel DevThomasJ.

Solving for X requires a matrix transpose X to Z of all input matrices after copying them to the device. It also requires a matrix transpose Z to X of the Res matrix before copying this matrix back to the host.

### 1.6.3 Killing an instance

A Thomas instance is killed by calling Thomas::KillInstance and passing the CudaID as parameter. KillInstance should be executed when the ModuleCuda instance is killed. It will clean up all host and device resources used by the Thomas algorithm in CUDA.

---

\(^1\) Wielen, J.P. van der (2011) *Hydro-dynamics in CUDA – Programming Guide*
1.7 FORTRAN – CUDA binding

CUDA is an extension to C, and thus the CUDA kernels are written in C / C++. This requires that C methods can be called from FORTRAN. The best option would be to call C++ methods from FORTRAN; however this is currently impossible without a wrapper.

A new module was added to MOHIDBase: ModuleCuda. This module contains the bindings to all C methods that are used in the CudaWrapper and the CudaThomas project.

Binding a C method in FORTRAN requires using an interface. Below an example is shown of a C binding.

```
interface InitializeThomas_C
  subroutine InitializeThomas_C(ObjCudaID, Size) bind(c, name="InitializeThomas_C")
    use, intrinsic :: Iso_C_Binding
    ! import is needed to know the T_Size3D type within the interface
    import :: T_Size3D
    implicit none
    integer(C_INT)                  :: ObjCudaID
    type(T_Size3D)                  :: Size
  end subroutine InitializeThomas_C
end interface InitializeThomas_C
```

Table 1-2: Interface for C binding in FORTRAN

The `InitializeThomas_C` method initializes a CudaThomas object with ID ObjCudaID.

The `bind` command is used to bind the method to a C method with name `name` (case sensitive). It should be noted that the calling convention in the C / C++ project should be `__cdecl`, otherwise the linker will not be able to recognize the binding.

```
use, intrinsic :: Iso_C_Binding is needed to recognize the bind command.
```

Values are always given as pointers to the C code. See the C declaration of the `InitializeThomas_C` method below. The `objCudaID` integer is received as a pointer to an integer. Also the struct size is received as a pointer to a struct.

```
void InitializeThomas_C(int *objCudaID, T_Size3D *size);
```

If a struct is given as parameter, the struct should be imported into the interface with the `import` keyword. The struct should also be defined on the C side.

The integer `ObjCudaID` must be explicitly declared as a C_INT integer. A list of all intrinsic types for the Iso_C_Binding can be found on the Intel website.

1.7.1 Initializing CUDA and Thomas

The FORTRAN methods for initializing CUDA and Thomas are both placed in ModuleCuda for now. The initialization and solve methods for Thomas can be moved to ModuleThomas (this module does not exist yet), but for the proof of concept it is easier to use ModuleCuda.

---

**Figure 1-1** contains the process flow for initializing CUDA. In MohidWater CUDA is initialized in ModuleModel. Each ModuleModel instance that is created gets one instance of ModuleCuda, according to the logic of MOHID. ConstructCudaBinding_C does nothing more than checking of the GPU has been initialized for the process of the current application. The GPU needs to be initialized only once for an application, not for every ModuleCuda instance.

In **Figure 1-2** the process flow of initializing Thomas can be seen. InitializeThomas is called just after initializing CUDA; in ModuleModel, InitializeThomas_C calls CreateInstance of the C++ Thomas class. CreateInstance checks if a Thomas instance with the given ID already exists. If this is the case, nothing is done. If the instance does not exist yet, it is created. At this point memory for all necessary matrices is allocated on the GPU so the Solve methods don’t have to allocate and de-allocate this memory every time step.

KillCuda and KillThomas follow a similar process. KillThomas de-allocates all memory and KillCuda resets the device.
1.7.2 Flow of calling Thomas methods

Figure 1-3 shows the process flow of the Thomas algorithm as realized in CUDA. It has a binding between FORTRAN and C, as described in the previous paragraph.
MOHID has two methods for solving Thomas for a 3D grid: ThomasZ for the Z dimension and Thomas_3D for the X and Y dimension, both in ModuleFunctions. The use of these methods is maintained, only what happens inside the methods changes.

Take for example Thomas for the Z dimension. First ThomasZ (2 in FIGURE 1-3) is called, for example in the module ModuleHydroDynamics:

```
call THOMASZ(ILB, IUB, JLB, JUB, KLB, KUB, Me%THOMAS, Velocity_UV_New, Me%ObjCuda, .FALSE.)
```

The ID of the ModuleCuda object is given as parameter. In ThomasZ, SolveThomas in ModuleCuda is called (3):

```
call SolveThomas(CudaID, ILB, IUB, JLB, JUB, KLB, KUB, &
Thomas%COEF3%D, Thomas%COEF3%E, Thomas%COEF3%F, &
Thomas%TI, RES, 2)
```

In SolveThomas the correct ModuleCuda instance is selected, based on the CudaID. When this is done, the method SolveThomas_C (4) is called, which is only declared in an interface in the ModuleCuda:

```
call SolveThomas_C(ObjCudaID, ILB, IUB, JLB, JUB, KLB, KUB, D, E, F, TI, Res, Dim)
```

Again the CudaID is given as parameter, so the correct Thomas instance in C++ can be selected. The call to this method is translated by the compiler to a call to SolveThomas_C as declared and defined in C++. SolveThomas_C receives all inputs as pointers, and selects the correct Thomas instance (5):

```
Thomas *instance = Thomas::GetInstance(*cudaObjID);
```

Depending on the dimension Dim, the method then calls one of the following methods (6):

```
*zLBound, *zUBound, D, E, F, TI, Res);
*zLBound, *zUBound, D, E, F, TI, Res);
*zLBound, *zUBound, D, E, F, TI, Res);
```

The SolveThomas%dimension% method executes the process flow as described in §1.6.2.

### 1.8 Points of attention

There are some points of attention for using CUDA in MOHID:

- The Debug Double CUDA or Release Double CUDA configuration should be used. It is currently not possible to run both OpenMP and CUDA at the same time.
- The projects CudaWrapper and CudaThomas are static libraries. Therefore the runtime library is set to Multi-threaded (Debug) (/MT(d)), also to match the FORTRAN projects.
- The preprocessor directive _ENABLE_CUDA is used to create the module CUDA and to initialize the Thomas instance in C++. _ENABLE_CUDA is used to do correctness benchmarking for Thomas in both CUDA and FORTRAN code. Defining a preprocessor directive can be done in Configuration Properties -> Fortran -> General -> Preprocessor definitions. _ENABLE_CUDA must be set for all FORTRAN projects that use ModuleCuda.
• The preprocessor directive \_USE\_CUDA is used in FORTRAN code to define whether the Thomas algorithm should be run in CUDA or not. Define \_USE\_CUDA in all projects that use the directive. \_USE\_CUDA is currently used in MOHIDBase1, MOHIDBase2, MOHIDLand and MOHIDWater.

• The preprocessor directive \_USE\_PAGELOCKED determines whether page-locked memory allocation is used or not. \_ENABLE\_CUDA must be defined if \_USE\_PAGELOCKED is defined. \_USE\_PAGELOCKED must be defined for all projects that allocate memory for the Thomas algorithm. \_USE\_PAGELOCKED must also be defined for the CudaWrapper project. Page-locked memory can only be used in FORTRAN 2003 and higher because it requires an array bounds remapping that is not implemented in FORTRAN <= 95. If compiling with FORTRAN 95 or lower, the \_USE\_PAGELOCKED directive should not be defined.

• The executable projects that use CudaThomas should have CudaThomas as a dependency (currently MOHIDWater and MOHIDLand). The library projects do not need this dependency.

• Select the correct CUDA architecture / compute capability for the device where the code will run on. The Thomas algorithm can only run on a device with compute capability 1.3 or higher, due to the double precision floating point dependency. The architecture can be set in Configuration Properties -> CUDA C / C++ -> Device. A list of CUDA enabled GPU’s and their compute capability can be found on the NVIDIA website.

• Using intent(IN) in FORTRAN code will still send the value as a pointer to the C code, even though the value is expected not to be changed by the called method. Everything that is sent to a C method is a pointer.

1.9 Realized optimizations
General tests have shown the impact of several optimizations for CUDA code. These optimizations have been applied to the Thomas algorithm. The impact of each optimization for the Thomas algorithm specifically is researched in the test report for Thomas. The implementation of the optimizations is described in the following paragraphs.

1.9.1 Page-locked memory and memory alignment
Using page-locked memory can speed up the memory transfers between host and device up to 1.6x. FORTRAN does not have a default method to allocate page-locked memory, so this must be done with the CUDA API call cudaHostAlloc. Freeing this memory is done with cudaFreeHost. Another memory optimization is using aligned memory so memory coalescing is optimized. This is a very important optimization. The Thomas performance test showed that even code on the host executes faster if padded matrices are used.

Allocating memory for FORTRAN in CUDA C / C++ is not as easy as it seems. A generic method has been implemented in ModuleCuda to make allocating memory for a 3D matrix easier. Besides the usual declaration of a 3D matrix in FORTRAN, a type(C\_PTR) variable has to be declared.

Take for example the D coefficient of the Thomas algorithm. It is declared as follows, with or without using page-locked memory:

\[
\text{D coefficient:} \quad \text{with page-locked memory:} \quad \text{D} : \text{type}(C\_\text{PTR})
\]

\[
\text{D coefficient:} \quad \text{without page-locked memory:} \quad \text{D} : \text{C}\_\text{PTR}
\]

---

4 Wielen, J.P. van der (2011) Hydro-dynamics in CUDA – Test report
5 Wielen, J.P. van der (2011) MOHID in CUDA – Test report
6 Wielen, J.P. van der (2011) Hydro-dynamics in CUDA – Test report, ch.2
7 Wielen, J.P. van der (2011) MOHID in CUDA – Test report, ch.3.4.4
To use page-locked memory the following pointer is declared:

```fortran
real, dimension (:,:, :) , pointer :: D
```

The C_PTR is a pointer that can be used in C. A normal FORTRAN pointer cannot be used, since its architecture is very different from a C pointer. To allocate memory for D, the following ModuleCuda subroutine is called:

```fortran
call Alloc3DPageLocked(Me%ObjCuda, DPtr, D, IUB + 1, JUB + 1, KUB + 1)
```

IUB, JUB and KUB are the upper bounds of the model geometry. These bounds are inclusive, and since MOHID uses matrices with a 0-index base, 1 has to be added. Alloc3DPageLocked calls the C method which allocates the page-locked memory and binds it to the given C pointer. After this, the method c_f_pointer binds the TmpArr to the C pointer. By default FORTRAN uses matrices with a 1-index base, and c_f_pointer follows this convention. The 1-index based matrix TmpArr has to be remapped to the 0-index based matrix Arr. It should be noted that this remapping is only supported in FORTRAN 2003 and higher.

```fortran
subroutine Alloc3DPageLocked (ObjCudaID, Ptr, Arr, XDim, YDim, ZDim)
    real(C_DOUBLE), dimension(:, :, :), pointer :: TmpArr
    call Alloc3DPageLocked_C(Ptr, XDim, YDim, ZDim)
    call c_f_pointer(Ptr, TmpArr, [XDim, YDim, ZDim])
    Arr(0:, 0:, 0:) => TmpArr
end subroutine
```

The XDim that is given to Alloc3DPageLocked_C is altered in C++ to create a width that ensures the use of padded memory. Therefore with page-locked memory a larger array is allocated than is actually needed.

The used memory can be free by calling FreePageLocked in ModuleCuda:

```fortran
call FreePageLocked(Me%ObjCuda, DPtr, D)
```

Both the pointer and the matrix are given as parameters. The pointer is used to free the memory and the matrix is nullified.

Altogether the code for using an arbitrary matrix Matrix looks as shown in Table 1-3.

```fortran
! Declarations
real, dimension (:,:, :) , pointer :: Matrix
type(C_PTR) :: MatrixPtr

! Inside some Construct subroutine
call Alloc3DPageLocked(Me%ObjCuda, MatrixPtr, Matrix, xSize, ySize, zSize)

! Do something with Matrix
...

! Inside some Kill subroutine
call FreePageLocked(Me%ObjCuda, MatrixPtr, Matrix)
```

Table 1-3 - Code for allocating page-locked and padded memory
1.9.2  Matrix transposing for coalesced memory
Memory alignment alone is not always enough to ensure coalesced memory accesses. Memory coalescing is relatively easy for the Y and Z dimension, but when solving for the X dimension, matrix transposing is needed.

In short memory coalescing means that accessing contiguous memory addresses at the same time is faster than doing random memory accesses. Matrices are stored in an X-minor manner in MOHID. This means that contiguous memory accesses along the X dimension are the fastest. i.e. thread [0,0] accesses [X = 0, Y = 0], thread [1,0] accesses [X = 1, Y = 0] at the same time for Z = 0 to Z = n – 1 if solving for the Z dimension. This can be realized if solving for the Y and Z dimension, but not when solving for the X dimension. Thread [0, 0] needs to access [Z = 0, X = 0] and thread [1, 0] needs to access to [Z = 1, Y = 1] for X = 0 to X = n – 1, which means memory accesses are not coalesced. This is solved by transposing the matrix from X-minor to Z-minor. Matrix transposing and memory coalescing are described extensively and more visually in the programming guide\(^8\).

1.9.3  Registers instead of global memory
Using registers to store global memory is beneficial if the same memory is accessed multiple times by the same thread. This is the case in the Thomas algorithm, so the algorithm has been optimized by using registers. The test report shows that the benefit of using registers is relatively small in this case, because the compiler does a lot of optimization if only global memory is used. i.e., the compiler tries to figure out which global memory is accessed more often and stores it in registers. In a more complicated algorithm the compiler might not be able to figure out the dependencies of memory use, so it might not always do these optimizations. Therefore it is a good habit to use registers where possible. What should be taken into account is that the more registers are used the less threads can run simultaneously.

1.9.4  Concurrent copy and execution
Concurrent copy and execution means that a memory transfer between host and device is performed while a kernel is executed on the device. This is done in the case of the matrix transposing for dimension X. D, E, F, TI and Res all have to be copied and transposed. When D has been copied, the transfer for E can be started and Res can be transposed at the same time. Using this technique minimizes the overhead of matrix transposing. Streams are used to enable concurrent copy and execution\(^9\).

---

\(^8\) Wielen, J.P. van der (2011) *Hydro-dynamics in CUDA – Programming Guide*

2 Conclusion
This document described the process flow of using CUDA in FORTRAN and in MOHID specifically.

Chapter 1 showed that it is possible to use CUDA in FORTRAN, a C binding needs to be established. The flow appears to be complex due to using three languages, but it is doable.

CudaWrapper and ModuleCuda should make it easier to use CUDA in MOHID, since a number of generic methods have been developed. For example allocating and de-allocating page-locked memory is a matter of calling two ModuleCuda methods instead of calling allocate and de-allocate.

Considerations are the fact that using page-locked memory requires FORTRAN 2003, and running the Thomas algorithm in CUDA requires a graphics card with compute capability 1.3 or higher.